

Frequency Distribution of Error Messages

David Pritchard

Center for Education in Math and Computing, University of Waterloo, Canada *
dagpritchard@uwaterloo.ca

Abstract

Which programming error messages are the most common? We investigate this question, motivated by writing error explanations for novices. We consider large data sets in Python and Java that include both syntax and run-time errors. In both data sets, after grouping essentially identical messages, the error message frequencies empirically resemble Zipf-Mandelbrot distributions. We use a maximum-likelihood approach to fit the distribution parameters. This gives one possible way to contrast languages or compilers quantitatively.

Categories and Subject Descriptors D.3.4. [Programming Language Processors]: Compilers, Run-time environments

Keywords Error messages, empirical analysis, usability, education.

1. Introduction

This work started as an offshoot of Computer Science Circles (CS Circles) [33, 34], a website with 30 lessons and 100 exercises teaching introductory programming in Python. It contains a system where students can ask for help if they are stuck on a programming exercise. Often, students reported being stuck because they could not comprehend an error message, asking for a better explanation of what the compiler/runtime was trying to say. E.g., the message

```
SyntaxError: can't assign to function call
```

might not be understood by a novice who wrote `sqrt(y)=x`.

Motivated by this, we decided to systematically improve the error messages that students received. There is copious literature on writing good error messages [14, 25, 26, 29,

39, 40], but how can this advice be incorporated into the programming ecosystem? One approach would be making upstream improvements to the compiler/runtime, but this can take a long time, and not all audiences would appreciate the changes that would most benefit novices. A second approach would be to write a tool that analyzes code from scratch, looking for common syntactic bugs or likely semantic mistakes. The literature includes many such tools: see `checkstyle`, `findbugs` and [10, 15, 20, 22, 23, 36].

We chose a more lightweight approach: augmenting the normal error messages with additional explanations. To wit, we compile and execute the code as usual, and then add a beginner-appropriate elaboration of the resulting error message, implemented by rendering the normal error with a clickable pop-up link to the explanation. This augmenting-explanation approach has been previously used on a small scale with Java compiler errors [6, §5.2], Python runtime errors [13, §5.2.1], and C++ STL compiler errors [41].

It has long been observed that “a few types of errors account for most occurrences” [35], see also [7]. In order to make sure that a small number of explanations would be useful as often as possible, we had to answer the following question: *what error messages are the most common?* Counting error message frequencies has a long history, starting from assembler [4, 29] and SP/k [14], with renewed interest more recently, using much larger data sets [1, 7, 17, 18, 38].

Using with the history of all previous submissions, we determined the essentially distinct error messages and their frequencies (see Section 2), available online at <http://daveagg.github.io/errors>. We wrote explanations for the 36 most common messages. Regular expressions were used to aid the implementation. At the most basic level, some errors were made more readable by elaborating them into a full paragraph of text rather than a one-line message. Some explanations include concrete examples of code that causes the same error message, and a description of how to fix it. See [14, 25, 26, 29, 39, 40] for advice on writing error messages. Given a large data set, the work involved in this group-and-explain approach is modest and not technically challenging, so we would recommend it in any beginner-facing system. Moreover, in internationalized settings, one can then add explanations in other languages (this has been implemented in CS Circles’ Lithuanian translation).

* Work started while located at Princeton University and completed at U. Southern California. Currently located at Google Los Angeles.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLATEAU '15, Month d–d, 20yy, City, ST, Country.
Copyright © 2015 ACM 978-1-xxxx-xxxx-n/yy/mm...\$15.00.
<http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn>

This paper compares and contrasts the most common error messages in CS Circles with those in another programming language. The Blackbox project [1, 21] is a large-scale data collection-and-sharing project using BlueJ, a Java programming environment oriented at beginners. We obtained the error messages from all recorded compilation and execution events, grouping and counting the essentially different messages like we did for the Python data set. Comparing the two data sets, we found that both error message frequency distributions resembled the same family of distributions, the Zipf-Mandelbrot distribution [24]. For these data sets, this means that for any integer k , the frequency of the k th most common error is approximately proportional to $1/(k + t)^\gamma$ where t and γ are parameters of the data set. In order to determine the best values for these parameters, we propose using a simple maximum-likelihood approach.

1.1 Discussion and Other Related Work

Orthogonal to purely quantitative analysis, a large body of work focuses on manual categorization of errors. This allows researchers to get more accurate results, and to precisely understand the psychological state of the user, rather than focus on the compiler-generated error messages themselves. Good reasons for doing this include that “A single error may, in different context, produce different diagnostic messages” and that “The same diagnostic message may be produced by entirely different and distinct errors,” see [27]. This analysis also helps measure whether a compiler’s error messages are appropriate (e.g., see [19, 35]). This analysis is important for compiler designers, language designers and educational research, but it is not our focus.

The comparison of error message frequencies between different languages raises many interesting open-ended questions. Even within the same language, some compilers are significantly better or worse than others; see Brown’s amusing crowdsourcing of Pascal error messages [2, 3] as well as [31, 40]. One way to view different error message distributions is to imagine the extremes: the worst possible language would only ever say “?” without elaborating (this has been formally evaluated, see [39]), while the best possible language would, like a human tutor, always give a perfectly adapted explanation. The exponent γ in our work is one way to measure where a language sits between these extremes. However, simpler measures such as entropy could also be used. Also, a single quantitative measure should not be treated as paramount without context. When comparing languages/compilers (e.g., [28]), statistical fitness is less important than overall usability, including measures like time between errors and time to achieve user goals.

A notable alternative approach to improving student feedback based on large-scale data, rather than focusing on error messages, is the HelpMeOut system [12], which uses a detailed repository of past student work sessions to find old errors similar to new ones and make suggestions of how to fix them.

To our knowledge, this paper is the first one to examine any link between programming error messages and statistical distributions. The special case $t = 0$ of the Zipf-Mandelbrot distribution is known as the power law distribution. It arises empirically in data sets such as the frequency of distinct words in books, of links in webpages, and of citations in literature. Caveats apply here [5, 11, 30], including: that generative explanations of how these distributions could arise are tenuous; that near-power-law data sets may be even closer still to other distributions; and that analyzing such data sets has common pitfalls like using linear regression. Another caveat for our work is that the distributions of error messages will depend on the nature of the users, and the kind of setting in which the work is collected. In our case, both data sets come from a very large, open project intended for beginners. We anticipate that a data set where students only work on a fixed set of exercises could be skewed in some way, but both BlueJ and the CS Circles “console” allow students to do any sort of open-ended programming.

See [32] for discussion of power laws in runtime object-reference graphs of industry-scale computer programs.

2. Data Sets

Our first data set is the Python corpus from CS Circles. Amongst the first 1.6 million code submissions, about 640000 resulted in an error. Our second data set is the Java corpus from BlueJ Blackbox. We specifically considered the “compile” events, of which there were about 8 million, half of which produced an error, and the “invoke” events, of which there were about 5 million, about 260000 of which produced a syntax error and 180000 of which produced a run-time error. We did not include the codepad or unit test events, both of which are an order of magnitude smaller.

In both cases, following [31], we only counted the first error message. This tends to be the most accurate error (since a syntax error can cause new valid parts of the program below to be reported as errors) and it is also the error that the programmer is most likely to pay attention to and fix first. Moreover, CS Circles only shows the first error message in its user interface; and even for a UI like BlueJ that shows multiple errors, beginner students often (by habit or by instruction) fix only one at a time and then recompile/re-run.¹

After obtaining these raw data sets of hundreds of thousands of error messages, we had to count how many times each distinct message occurred. It is necessary to “sanitize” the data by removing parts that pertained to specifics of user code rather than the kind of error. For instance, `NameError: name 'x' is not defined` should be understood by our system to be essentially the same error as `NameError:`

¹ It would not be invalid to investigate data sets where all errors are reported and counted, but a worry is that it might say more about the statistics of chain effects in syntax errors and less about the actual underlying bugs. Two other strategies, “count-all” and “count-distinct,” are used in [38], though their study participants were professionals and not novices.

name 'sum' is not defined so that the same explanation will appear in either case. The sanitization was an iterative process. Simple heuristics handled most cases correctly, and in total we needed about 20 sanitization rules for Python and 50 for Java, implemented using regular expressions.

There is a question of how far one should sanitize. Should these two error messages be considered the same?

```
RuntimeError: maximum recursion depth exceeded
while getting the repr of a list
RuntimeError: maximum recursion depth exceeded
while getting the repr of a tuple
```

Overall we tended to use fewer sanitization rules rather than more (considering the above to be different); a similar approach was used in [38]. Conceptually, to fix a single objective goal for sanitization, we imagined that each category should uniquely correspond to a single line of source code of the compiler/runtime where the error is first detected.

Another step in sanitization was to remove any non-English error messages, to avoid inadvertently seeing the same patterns repeated in multiple languages, which might affect the results. This was done by removing all messages with non-ASCII characters, and manual filtering.

2.1 Overview of Data Sets

The Python data set yielded 309710 syntax errors and 333538 compile-time errors. The Java data set yielded 4002822 compile-time errors and 129650 run-time errors. Note that the Java data set has a much smaller proportion of run-time errors than Python (only about 3% rather than almost half). But to a degree, this difference is inherent in the language, since many errors that would occur at compile-time in Java's strict typing-and-scoping system are not encountered until run-time in Python.

After sanitization and grouping, the Python data set yielded 283 distinct error messages. Of these, 17 occurred exactly twice and 42 occurred only once (for example, `ValueError: Format specifier missing precision` and `SyntaxError: can't assign to Ellipsis`). The Java data set yielded 572 error messages in total; 65 occurred exactly twice and 127 occurred only once (for example, `com.vmware.vim25.InvalidArgument` and `cannot create array with type arguments`).

Errors are not completely parallel for both languages. For example, Java allows function overloading, i.e. two functions with distinct signatures but the same name. In Python, this must instead be implemented by a single function that takes different actions depending on the runtime number and type of its argument(s). It is the function's responsibility to generate the error message. It turns out that not all such functions generate identical messages and so the single Java error message `no suitable method found` corresponds to more than one distinct Python error message:

```
f argument must be a string or number, not T
and f arg 1 must be a type or tuple of types.
```

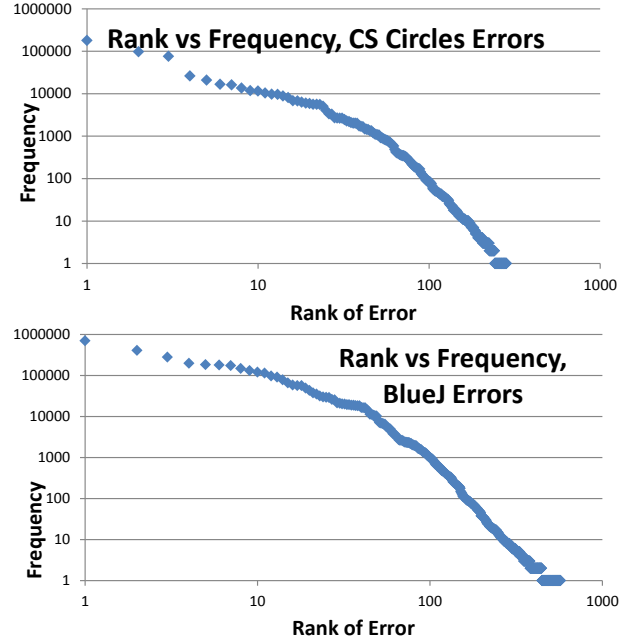


Figure 1. The two data sets for our study. The CS Circles data set is Python, while BlueJ is Java. The plots are log-log.

The 5 most common Python errors were:

```
179624 SyntaxError: invalid syntax
97186 NameError: name 'NAME' is not defined
76026 EOFError: EOF when reading a line
26097 SyntaxError: unexpected EOF while parsing
20758 IndentationError: unindent does not match
any outer indentation level
```

The 5 most common Java errors were:

```
702102 cannot find symbol - variable NAME
407776 ';' expected
280874 cannot find symbol - method NAME
197213 cannot find symbol - class NAME
183908 incompatible types
```

We plot both data sets in Figure 1. The x -axis measures the rank of each error message (with 1 being the most frequent) and the y -axis measures the number of times each error occurred. Using a logarithmic scale is necessary for the changes in the y -axis to be visible, and we also use a logarithmic scale for the x -axis. Notice that both data sets give rise to similar distributions; in the rest of the paper we will try to describe them in a common framework.

2.2 Notation

For any given data set, we will use N to denote the total number of errors logged, and M for the number of distinct error types. For example, the Python data set has $N = 643248$ and $M = 283$. Let F_k denote the number of times that the k -th-most common error occurred, e.g. $F_1 = 179624$ for Python. We will also write $F_{\max} := F_1$ as an alternate symbol for the same value, when we wish to emphasize that

$\#F^{-1}(f)$ where f is:	1	2	3	4	5	6
Python	42	17	19	13	6	4
Java	127	65	31	17	18	15

Table 1. Number of f -legomena in each data set.

it is the maximum frequency. The smallest frequency F_M is 1 for both of our data sets. In lexicography, the items occurring just once are known as the *hapax legomena* of the corpus. An f -legomenon is any error message that appears exactly f times. We will use the symbol

$$\#F^{-1}(f)$$

to denote the number of f -legomena. The first few counts of f -legomena in our data sets is listed in Table 1.

3. Power Law Distributions

When studying frequency counts of different objects, a discrete *power law* distribution is one in which the frequency F_k of the k th most common item is proportional to $1/k^\gamma$. As mentioned in the introduction, power law distributions provide good fits to many unrelated empirical distributions. A common example is that in the novel *Moby Dick*, the frequency of the k th-commonest word is approximately proportional to $1/k^{1.05}$. “Zipf’s law” is sometimes used as a synonym for the discrete power law, but sometimes also refers to the special case $F_k \propto 1/k$ where $\gamma = 1$. There is also a large body of work on continuous power laws, where one sorts items by some magnitude that takes on continuous values, and examines the relationship between rank and magnitude. See many examples of both types in [5].

Note that sanitization of error messages is particularly important because of the fact that many natural languages follow power-law curves. If we did absolutely no sanitization, then our error message distributions would have significant aspects determined by the frequency distribution of variable names chosen by users, and finding a power law describing the latter would be less surprising, given that natural language is already known to exhibit power law behaviour.

We now turn to analyzing our data sets from the power law perspective. Do they approximately satisfy a power law? This did not appear to be the case: a power law, when plotted on a log-log scale, should give a straight line, but it is clear from Figure 1 that this is not an accurate description of our data set.

3.1 Zipf-Mandelbrot Distributions

There is a generalization of power law distributions called the *Zipf-Mandelbrot* family of distributions. These distributions are defined, using two parameters t and γ , by

$$F_k \propto \frac{1}{(k+t)^\gamma}.$$

Such a sequence should appear linear on a log-log plot provided that along the x -axis, we plot the logarithmic positions of $(k+t)$ rather than of k . When we tested plotting these distributions in this modified way, for an appropriate value of t , we obtained a much more persuasive fit: in Figure 2, which has the shift $t = 60$, the points very nearly fall on a line. This shift was obtained by trial-and-error, and the line drawn in has slope $-\gamma = -6.3$. In the rest of the paper we aim to give a more principled way of estimating t and γ .

Is a Zipf-Mandelbrot distribution plausible? Here is one argument that, if we accept that power laws can arise in natural settings, that there is reason to suspect that Zipf-Mandelbrot laws can too. It is not meant to give an exhaustive explanation, just an argument for plausibility. Suppose we start with a power law, and then coalesce several items together. I.e., replace several distinct error messages with a single unified message having the sum of their frequencies. (In a list of English words, the analogy would be that a single word has multiple meanings.) The effects of this message-merging would be twofold: the resulting new message would be an outlier to the original power law curve; and the remaining data points, when plotted on a rank-frequency scale, would be shifted several positions to the left, i.e. they would follow a Zipf-Mandelbrot distribution instead of a power law distribution. This is indeed a plausible scenario for the Python data set! The most common error, `SyntaxError: invalid syntax`, is very generic. It can be obtained by writing two tokens in a row (such as forgetting a comma or quote marks), using an assignment statement in place of a conditional expression (such as using `if a=b:` instead of using `==`), by mismatching parentheses, etc.

3.2 Consequences of a Zipf-Mandelbrot model

What behaviour does a Zipf-Mandelbrot model predict? It postulates that there is some innate ordering of error messages, from most frequent to least frequent, so that the inherent probability F_ℓ^* of the ℓ th most frequent error message is proportional to $(\ell+t)^{-\gamma}$. The reason that we use the sub-

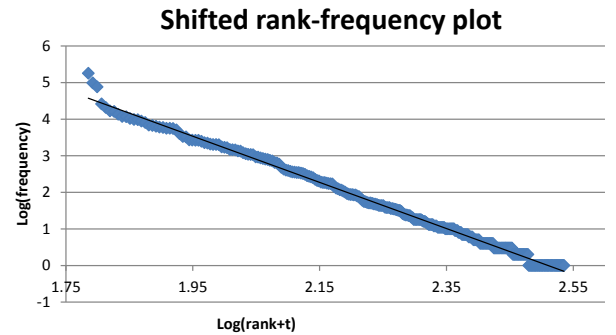


Figure 2. The Python data set with the (pre-logarithmic) x -axis shifted by $t = 60$, and a straight line with slope $-\gamma = -6.3$ that approximately fits most of the data.

script ℓ here is that our concrete data set arrives via sampling from the inherent distribution. So like a sampling error, the observed ordering of messages from most to least frequent is not necessarily the exact same as the innate ordering.

An interesting aspect of this model is that it assumes $F_\ell^* \propto (\ell + t)^{-\gamma}$ continues to hold for arbitrarily large ℓ . Can this be plausible: is the total number of possible errors infinite? We will accept this as a reasonable hypothesis, which if not literally true, could continue long enough to be consistent with the size of any measured data set, for the following reason, using Python as an example. The most common errors we see are the ones in the Python core code (the syntax errors, and runtime errors from the “builtins”). Less frequently we start to see errors from Python modules, such `ValueError: math domain error` within the `sqrt` function of the `math` module. While this is the only module taught on the site, users have occasionally submitted code using other common modules like `time`, `random` and `functools`, each of which comes with its own specific errors. Moving on, there would be errors from rarely-used modules, then even after this, modules that users may with more or less frequency import (or copy in) themselves. For example, we observed a `mainfile: error: must provide name of pdb file` error caused by someone who copied in a Python program for use with the X-PLOR biomolecular structure determination software [37]. The same phenomenon happens in the Java data set. So errors with arbitrarily small inherent frequency are not unreasonable despite the finite size of the languages.

4. Analysis

To fit our data to a Zipf-Mandelbrot distribution, several approaches are possible. For power laws, the naive approach, using a least-squares fit to a linear log-log plot (c.f. Figure 2) is known to introduce errors [11]. Rather, we will follow Newman [30], who considered maximum likelihood estimation methods for power laws. Some work will be needed to extend this to Zipf-Mandelbrot distributions.

The method in [30] involves a particular way of processing the data; let us mention the motivation. The direct approach to maximum likelihood estimation would be to determine the γ and t that maximize $\prod_k (C/(k+t)^\gamma)^{F_k}$ where C is the normalizing constant with $C \cdot \sum_{k=1}^{\infty} (k+t)^{-\gamma} = 1$. Izsák [16] suggests this. But trying this approach gives unsatisfactory results with any of our data sets — the curves produced fit the data very poorly except in the regime of F_1 . The calculation goes wrong because it is too heavily biased by the highest-frequency errors. (For Zipf-Mandelbrot in particular, if the argument in Section 3.1 were to be true, then it should be no surprise that fitting to F_1 would be problematic, since F_1 would be an outlier from the norm.) Also, the most likely fit entails that the innate order exactly matches the observed frequency-ordering of error messages [16], which is itself unlikely.

4.1 Probabilities of Frequencies

This motivates the maximum likelihood method on frequencies [5, 30]. It starts by taking a different view of the data set. Using the Python data set as a concrete example, we imagine the frequency vector $\mathbf{F} = (179624, 97186, \dots, 1, 1)$ itself as being an unordered set of M data points from a parameterized distribution — given a new error message, how frequent is it? This distribution-on-frequencies is a transformed version of the inherent distribution F^* , and also depends on the data set size. The goal, then, is to choose the parameters so as to maximize the likelihood of observing \mathbf{F} .

The analysis in [5, 30] primarily achieves rigor for continuous distributions. For discrete distributions, it turns out that the distribution-on-frequencies is actually given by another distribution which seems to have been first studied by Evert [8]. To describe it we recall the Γ function, which is the (shifted) analytic continuation of the factorial function, satisfying $\Gamma(n) = (n-1)!$ at positive integer values and $\Gamma(n) = (n-1)\Gamma(n-1)$ on its whole domain. The beta function, another standard function, is a continuous analogue of the binomial coefficient, defined by

$$B(x, y) := \Gamma(x)\Gamma(y)/\Gamma(x+y).$$

Then, finally, the *Evert distribution* is the frequency distribution, parameterized by one parameter α , defined by

$$\text{frequency of } f \propto B(f+1-\alpha, \alpha).$$

It is involved in our analysis for the following reason:

Proposition 1. *Suppose that we draw samples from a discrete Zipf-Mandelbrot distribution with parameters γ and t . If the number of samples is large, then for all small f , the expected number of f -legomena is proportional to $B(f+1-\alpha, \alpha)$ where $\alpha = 1 + 1/\gamma$.*

Paraphrasing, this says that the distribution-on-frequencies for a discrete Zipf-Mandelbrot distribution is the Evert distribution. This result was obtained by Evert [8] though he expressed it in terms of “type density functions.” We reprove it in Appendix A.

We remark that the proof of Proposition 1 remains valid even if the discrete Zipf-Mandelbrot distribution is perturbed by altering some of the highest probabilities, which ensures that it is still valid even if outliers à la Section 3.1 occur.

4.1.1 Remarks

In [5, 30], the focus of the analysis is on continuous power law distributions, and for that, the analogue of Proposition 1 is to use a simple power law with exponent α instead of an Evert distribution. Though the Evert distribution is not mentioned in [30], it is remarked that the another distribution, the Yule distribution, is an “an alternative and often more convenient form” of the discrete power law. These conveniences are mathematical in nature: the normalizing constant, expectation, variance, and moments of the Yule distribution have

nicer closed forms than a pure power law. (And it is reasonable to use in power law analysis because up to a scaling factor, the Yule distribution becomes a discrete power law in the limit.) These conveniences holds for the Evert distribution too, since Evert and Yule differ only by a shift. For instance, our fitting code utilizes the identity $\sum_{f=1}^{F_{\max}} B(f+1-\alpha, \alpha) = B(2-\alpha, \alpha-1) - B(F_{\max}+2-\alpha, \alpha-1)$.

4.2 Maximum Likelihood

The Evert distribution allows us to compute the most likely value of α for the collection of frequencies \mathbf{F} . Writing E_f^α for $B(f+1-\alpha, \alpha)$, and C^α for the normalizing constant with $C^\alpha \cdot \sum_{f=1}^{F_{\max}} E_f^\alpha = 1$, we seek the α that maximizes

$$\prod_{k=1}^M C^\alpha \cdot E_{F_k}^\alpha.$$

This can be determined numerically using binary search, using logarithms since the numbers involved are very small. Then we determine the parameter γ using $\gamma = 1/(\alpha - 1)$.

The only remaining issue is how to determine the value of the shift parameter t that has maximum likelihood. Proposition 1 does not help since t plays no role in its conclusion. (The reason for this apparent paradox is that the approximation guarantee of Proposition 1 is only valid for small frequencies.) Nonetheless, we can determine a value for t using some ideas from the analysis of the continuous case [5, 30].²

Proposition 2. *Let $\alpha = 1 + 1/\gamma$. Suppose we draw a sample from the bounded continuous power-law distribution with exponent $-\alpha$ and domain $(1, (\frac{t}{t+M+1})^{-\gamma})$. Then the $(M+1)$ -quantiles of this random variable are proportional to $(t+1)^{-\gamma}, (t+2)^{-\gamma}, \dots, (t+M)^{-\gamma}$. Furthermore, the choice of t that maximizes the likelihood of observing \mathbf{F} is $t = (M+1)/(F_{\max}^{1/\gamma} - 1)$.*

The first conclusion says that a “typical” draw of M items from this continuous distribution-on-“frequencies” is a model for the Zipf-Mandelbrot distribution. The second conclusion gives us the rule that we use to compute t in our statistical fitting. We prove the proposition in Appendix B.

5. Fitting the Data

In Evert’s paper [8], rather than using maximum-likelihood, he proposes estimating α using a Chi-squared test on the first few $\#F^{-1}(1), \#F^{-1}(2), \dots$ values. This approach is implemented by the R library `zipfR` of Evert and Baroni [9].

We fit our data sets to the Zipf-Mandelbrot family of distributions, using both the Chi-squared approach, and the maximum-likelihood method of Propositions 1 and 2 (implemented in Maple). The results of the fitting are shown in

²The authors of [5, 30] note that the continuous model reasonably resembles the discrete model when thinking about larger frequencies; the smallest continuous variables are the ones that would have to be distorted the most in order to become quantized. Thus, the inaccuracies of Proposition 2 are complementary to those of Proposition 1.

	Max. likelihood	χ^2 min.
Java	$\alpha = 1.216, t = 33.1$	$\alpha = 1.225, t = 25.7$
Python	$\alpha = 1.165, t = 44.7$	$\alpha = 1.143, t = 65.7$
Python ^{w/o}	$\alpha = 1.131, t = 99.8$	$\alpha = 1.133, t = 92.9$

Table 2. Results of fitting our data sets to Zipf-Mandelbrot distributions with both methods. Python^{w/o} indicates the Python data set with the 3 commonest messages removed.

Shifted log-log plot of CS Circles error frequencies (with 3 outliers removed)

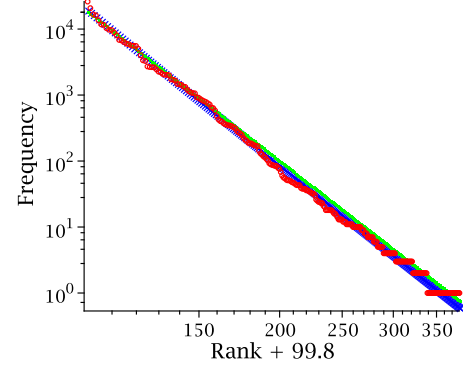


Figure 3. Plot of the Python data set on shifted log-log axes, with shift t from maximum likelihood estimation. Red: observed frequencies; blue: Chi-squared fitted Zipf-Mandelbrot distribution; green: maximum likelihood fitted Zipf-Mandelbrot distribution.

Shifted log-log plot of BlueJ error frequencies

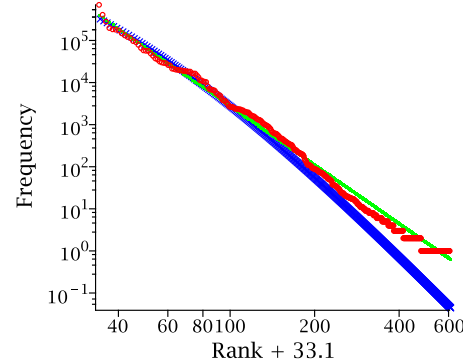


Figure 4. Plot of the Java data set, analogous to Figure 3.

Table 2. The fit for the Python data set improved greatly by treating the three most common errors as outliers (c.f. Figure 1). In Figures 3 and 4 we show shifted log-log plots of the observed fits (the Python plot omits the outliers). For the Python-without-outliers data set, both methods give a good fit. For the Java data set, the maximum-likelihood method gives a significantly better fit than the Chi-squared method.

6. Future Work

A few very short questions for future work are: (1) can the good fit be replicated in other Java/Python systems? (2) if so, what properties of the user base or programming ecosystem affect the α and t parameters? (3) do error messages in other languages also follow a Zipf-Mandelbrot distribution?

In the context of the hypothetical extreme languages of Section 1.1, Python's slightly smaller value of α suggests that it tends to give more distinctive error messages. Is it actually giving more information in its errors? Could it alternatively be explained due to artefacts like the non-parallelism mentioned in Section 2.1?

It would be interesting to re-analyze the discrete data sets in [30] using the Evert maximum likelihood method. Specifically, this could be done for the data sets for word frequency, web hits, telephone calls, and citations, which are discrete distributions coming from a population that is large enough to be effectively infinite. Additionally, it would be interesting to apply the Kolmogorov-Smirnov test suggested in [30] to the Evert maximum likelihood method, to be more rigorous in our approach.

From a more practical perspective, it would be not hard, and of a great potential benefit, to release a systematic data set of good beginner-friendly explanations of the top errors in different programming languages. Further work could try to quantify if this improves the ability of beginner students to program independently.

Acknowledgment

We thank the SIGCSE Special Projects committee, whose Summer 2013 grant for CS Circles provided funding when this work was initiated [34], and the Blackbox project for their work on providing accessible huge data sets. We thank undergraduate assistants Ayomikun (George) Okeowo and Pallavi Koppol for their work on sanitizing and writing explanations. Thanks to Jurgis Pralgauskis for translating the Python explanations to Lithuanian. We also thank the PLATEAU referees for their helpful suggestions.

References

- [1] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting. Blackbox: A large scale repository of novice programmers' activity. In *Proc. 45th SIGCSE*, pages 223–228, 2014.
- [2] P. Brown. 'My system gives excellent error messages'—or does it? *Software: Practice and Experience*, 12(1):91–94, 1982.
- [3] P. J. Brown. Error messages: the neglected area of the man/machine interface. *Comm. ACM*, 26(4):246–249, 1983.
- [4] J. M. Chabert and T. Higginbotham. An investigation of novice programmer errors in IBM 370 (OS) assembly language. In *Proc. 14th ACM-SE Southeast Regional Conference*, pages 319–323, 1976.
- [5] A. Clauset, C. R. Shalizi, and M. E. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.
- [6] N. J. Coull. *SNOOPIE: development of a learning support tool for novice programmers within a conceptual framework*. PhD thesis, University of St Andrews, 2008.
- [7] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proc. 17th ITiCSE*, pages 75–80, 2012.
- [8] S. Evert. A simple LNRE model for random character sequences. In *Proc. 7th JADT*, 2004.
- [9] S. Evert and M. Baroni. zipfR: Word frequency distributions in R. In *Proc. 45th Ann. Meeting ACL*, pages 29–32, 2007.
- [10] T. Flowers, C. Carver, J. Jackson, et al. Empowering students and building confidence in novice programmers through Gauntlet. In *Proc. 34th FIE*, pages T3H 10–13, 2004.
- [11] M. L. Goldstein, S. A. Morris, and G. G. Yen. Problems with fitting to the power-law distribution. *Euro. Phys. J. B-Condensed Matter & Complex Syst.*, 41(2):255–258, 2004.
- [12] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proc. 28th SIGCHI*, pages 1019–1028, 2010.
- [13] A. J. Hartz. *CAT-SOOP: A tool for automatic collection and assessment of homework exercises*. PhD thesis, Massachusetts Institute of Technology, 2012.
- [14] J. J. Horning. What the compiler should tell the user. In Brauer, F.L. et al., editor, *Compiler Construction*, volume 21 of *Lecture Notes in Computer Science*, pages 525–548. 1974.
- [15] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.
- [16] F. Izsák. Maximum likelihood estimation for constrained parameters of multinomial distributions — application to Zipf-Mandelbrot models. *Comp. statistics & data analysis*, 51(3): 1575–1583, 2006.
- [17] J. Jackson, M. Cobb, and C. Carver. Identifying top Java errors for novice programmers. In *Proc. 35th FIE*, pages T4C 24–27, 2005.
- [18] M. C. Jadud. A first look at novice compilation behaviour using BlueJ. *Comp. Sci. Ed.*, 15(1):25–40, 2005.
- [19] W. L. Johnson. Understanding and debugging novice programs. *Artificial Intelligence*, 42(1):51–97, 1990.
- [20] W. L. Johnson and E. Soloway. PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, SE-11(3):267–275, 1985.
- [21] M. Kölling and I. Utting. Building an open, large-scale research data repository of initial programming student behaviour. In *Proc. 43rd SIGCSE*, pages 323–324, 2012.
- [22] B. Lang. Teaching new programmers: a Java tool set as a student teaching aid. In *Proc. 1st PPPJ*, pages 95–100, 2002.
- [23] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. *ACM SIGPLAN Notices*, 42(6):425–434, 2007.
- [24] B. Mandelbrot. An informational theory of the statistical structure of language. *Comm. theory*, 84:486–502, 1953.

- [25] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proc. 42nd SIGCSE*, pages 499–504, 2011.
- [26] G. Marceau, K. Fisler, and S. Krishnamurthi. Mind your language: on novices’ interactions with error messages. In *Proc. 10th SIGPLAN*, pages 3–18, 2011.
- [27] D. McCall and M. Kolling. Meaningful categorisation of novice programmer errors. In *Proc. 44th FIE*, pages 1–8, 2014.
- [28] L. McIver. The effect of programming language on error rates of novice programmers. In *Proc. 12th PPIG Workshop*, pages 181–192, 2000.
- [29] P. G. Moulton and M. E. Muller. DITRAN – a compiler emphasizing diagnostics. *Commun. ACM*, 10(1):45–52, 1967.
- [30] M. E. J. Newman. Power laws, Pareto distributions and Zipf’s law. *Contemporary Physics*, 46(5):323–351, 2005.
- [31] M.-H. Nienaltowski, M. Pedroni, and B. Meyer. Compiler error messages: What can help novices? *ACM SIGCSE Bulletin*, 40(1):168–172, 2008.
- [32] A. Potanin, J. Noble, M. Freen, and R. Biddle. Scale-free geometry in OO programs. *Comm. ACM*, 48(5):99–103, 2005.
- [33] D. Pritchard and T. Vasiga. CS Circles: an in-browser Python course for beginners. In *Proc. 44th SIGCSE*, pages 591–596, 2013.
- [34] D. Pritchard, S. Graham, and T. Vasiga. The state of CS Circles: Open source and outreach with an introductory Python website (Poster). In *Proc. 46th SIGCSE*, page 688, 2015.
- [35] G. D. Ripley and F. C. Druseikis. A statistical analysis of syntax errors. *Computer Languages*, 3(4):227–240, 1978.
- [36] T. Schorsch. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. *ACM SIGCSE Bulletin*, 27(1):168–172, 1995.
- [37] C. D. Schwieters, J. J. Kuszewski, N. Tjandra, and G. Marius Clore. The Xplor-NIH NMR molecular structure determination package. *J. Magnetic Resonance*, 160(1):65–73, 2003.
- [38] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers’ build errors: A case study (at Google). In *Proc. 36th ICSE*, pages 724–734, 2014.
- [39] B. Shneiderman. Designing computer system messages. *Communications of the ACM*, 25(9):610–611, 1982.
- [40] V. J. Traver. On compiler error messages: what they say and what they mean. *Adv. Human-Computer Interaction*, 2010.
- [41] L. Zolman. STLfilt: An STL error message decryptor for C++, 2005. <http://www.bdsoft.com/tools/stlfilt.html>.

A. Proof of Proposition 1

Fix a constant f and consider N as growing to infinity. By linearity of expectation, the expected number $E[\#F^{-1}(f)]$ of f -legomena is equal to the sum, over all ℓ , of the probability that word ℓ occurs exactly f times in our sample. For large N , any word with frequency bigger than a constant has vanishingly small probability of occurring only f times. So for a word that may become an f -legomenon, its

number of occurrences is well-approximated by a Poisson random variable, since it is a sum of many Bernoulli random variables, each with a small individual expectation. The expected number of occurrences of the ℓ -th most common word is $NC(\ell + t)^{-\gamma}$, so the number of times we observe it is a Poisson variable with expectation $NC(\ell + t)^{-\gamma}$.

This means that for any constant f , by the definition of a Poisson variable,

$$\Pr[\text{word } \ell \text{ appears exactly } f \text{ times}] = \frac{(NC(\ell + t)^{-\gamma})^f}{f! \exp(NC(\ell + t)^{-\gamma})}.$$

Thus, the expected number of words appearing f times is

$$E[\#F^{-1}(f)] = \sum_{\ell=1}^{\infty} \frac{(NC(\ell + t)^{-\gamma})^f}{f! \exp(NC(\ell + t)^{-\gamma})}.$$

We approximate this infinite sum with the infinite integral

$$E[\#F^{-1}(f)] = \int_1^{\infty} \frac{(NC(x + t)^{-\gamma})^f}{f! \exp(NC(x + t)^{-\gamma})} dx.$$

To evaluate it, we substitute $y = NC(x + t)^{-\gamma}$, i.e. $x = (\frac{y}{NC})^{-1/\gamma} - t$ and so $dx = -\frac{1}{\gamma}(CN)^{1/\gamma}y^{-1-1/\gamma}dy$, giving

$$E[\#F^{-1}(f)] = \frac{(CN)^{1/\gamma}}{f! \gamma} \int_0^{NC^{-1}t^{-\gamma}} \frac{y^{f-\frac{1}{\gamma}-1}}{e^y} dy.$$

Again assuming N large, the above integral is well-approximated by replacing the upper bound by $+\infty$. Therefore, taking the terms that do not depend on f into the constant of proportionality, we find that

$$\begin{aligned} E[\#F^{-1}(f)] &\propto \frac{1}{f!} \int_0^{+\infty} \frac{y^{f-\frac{1}{\gamma}-1}}{e^y} dy \\ &= \frac{\Gamma(f - 1/\gamma)}{\Gamma(f + 1)} \\ &\propto B(f - 1/\gamma, 1 + 1/\gamma) = B(f + 1 - \alpha, \alpha). \end{aligned}$$

B. Proof of Proposition 2

Let $U(a, b)$ denote a random variable from the uniform distribution on (a, b) . Our starting observation is that the continuous power-law distribution with exponent $-\alpha$ and unbounded domain $(1, +\infty)$ is identical in distribution to $U(0, 1)^{-\gamma}$. See, for instance, [5, App. D].

Therefore, adding the bound to get the continuous power-law in the hypothesis of the theorem, said distribution is identical in distribution to $U(\frac{t}{t+M+1}, 1)^{-\gamma}$.

The $(M + 1)$ -quantiles of $U(\frac{t}{t+M+1}, 1)^{-\gamma}$ are $((t + k)/(t + M + 1))^{-\gamma}$ for $k = 1, \dots, M$, so the first conclusion follows.

Finally, the smaller the domain $(1, (\frac{t}{t+M+1})^{-\gamma})$, the larger the probability density function at the observed \mathbf{F} values, except that we need $(\frac{t}{t+M+1})^{-\gamma} \geq F_{\max}$ for F_{\max} to be observable at all. This proves the second conclusion.